

The Computational Universality of Metabolic Computing

Ricardo Henrique Gracini Guiraldelli and Vincenzo Manca
University of Verona

`{ricardo.guiraldelli,vincenzo.manca}@univr.it`

August 2, 2015

Abstract

System and synthetic biology are rapidly evolving systems, but both lack tools such as those used in engineering environments to shift their focus from the design of parts (details) to the design of systems (behaviors); to aggravate, there are insufficient theoretical justifications on the computational limits of biological systems. To diminish these deficiencies, we present theoretical results over the Turing-equivalence of metabolic systems, defines rules for translations of algorithms into metabolic P systems and presents a software tool to assist the task in an automatic way.

1 Introduction

There is an increasing interest, in the academic community, in the modeling of existing biological cells as well as synthesis of new ones. Systems biology and synthetic biology are parallelly developing themselves with great speed but, in some sense, apart and lacking mutual synergy. Nonetheless, a duality between these fields may be established and both may benefit of the same set of mathematical, computer science and engineering techniques to improve their analysis or design tasks; for instance, *molecular computation* [16, Section 20.7.5], in which the present work is certainly part of, is one of those tools.

Through the usage of the Church-Turing thesis [8, Section 5.1] [19, p. 183] [4, Section 2.5] and algorithmic construction, it will be shown that a particular kind of Metabolic P (MP) systems [9, Chapter 3] called *positively controlled MP systems (MPPC)* has the same computational power of a *register machine* and, hence, of a Turing machine, along with a demonstration of the equivalence relation between these models. Then, an outline of the software tool that converts a register machine to MPPC is presented. At last, there is a discussion on the importance of the results of this work as a connector between systems and synthetic biology and introduction of a computational perspective over these fields is stressed.

For this purpose, the present text is divided as follow: Section 2 and Section 3 introduces the bases of universal computing devices and MP systems; Sections 4 and 5 develop the equivalence of the both systems. Section 6 describes the developed tool and presents an example of translation of register

machine to MPPC grammar. Finally, at the last section, we discuss the advantages of the present formalism to both systems and synthetic biology.

2 Universal Computing Devices

The concept of (universal) *computability* is closely tied to concept of *algorithm* [8, p. 246] that, in turn, is tied with the concept of *Turing machine* [8, p. 246] [19, Definition 5.17], which may be considered the main computational device and stands out by its formal and construction simplicity.

Although simple, there are other computational devices equivalent to Turing machine that are more convenient to use at certain modeling occasions. Some examples of equivalent devices are *recursive functions*, *grammars* and *register machine*. In the present work, a particular kind of the latter will be used due to its simple instruction set and similarity to electronic computer architecture.

2.1 Register Machines

According to Minsky [12, Section 11.1], a *register machine* (or *program machine*) is composed of finite number of *registers* which has infinity capacity, a small set of *operations* over the registers and a hard coded *program*, which seems an indexed sequence of *instructions*. At last, an instruction is defined as a triple (*operation*, *register reference*, *list of instructions*).

The basic operations of a register machine must reflect those ones that defines recursive function, the (free) movement of the head of the Turing machine across the tape, a signal of end-of-computation and limit its operations to the set \mathbb{N} . Hence, the base register machine model in [12, Table 11.1-1] define four operations: *zero*, *successor*, *decrements or jump* and *halt*.

The present work, nevertheless, uses a variation [17, Section 4, p. 225] [11, p. 293] of the register machine defined by Minsky [12, Section 14.1] as following seen.

Definition 1 (Register Machine). *A register machine \mathcal{R} is a computational device defined as*

$$\mathcal{R} = (R, O, P)$$

where:

1. $R = \{R_1, R_2, \dots, R_n\}$ is a finite set of infinite capacity registers, with $n \in \mathbb{N}$;
2. $O = \{\text{INC}, \text{DEC}, \text{JNZ}, \text{HALT}\}$ is the set of operations;
3. $P = (I_1, I_2, \dots, I_n)$ is the program, with $n \in \mathbb{N}$.

The execution of the program P always start at the first instruction I_1 and procedures sequentially (unless for programmed execution re-route).

The instructions I of the register machine \mathcal{R} are special notations that conveniently name operations over addressed registers of \mathcal{R} according to Definition 2.

Definition 2 (Instructions). *Let the content of register R_i be equal to x . Then, it is possible to define the instructions I of the register machine \mathcal{R} as following:*

1. $\text{INC}(\mathbf{R}_i) = x + 1;$
2. $\text{DEC}(\mathbf{R}_i) = \begin{cases} x - 1 & , \text{ if } x > 0 \\ 0 & , \text{ otherwise} \end{cases};$
3. $\text{JNZ}(\mathbf{R}_i, \mathbf{I}_j)$ change the execution flow of \mathcal{R} setting \mathbf{I}_j as the next instruction to be executed in case $x > 0$; otherwise, the execution flow keeps sequential;
4. **HALT** ends the computation of \mathcal{R} .

A register machine, as seen above, is a simple but powerful computational device easily translate to digital computer architecture. Now it is time to examine that one inspired by cell metabolism, the *MP systems*.

3 Metabolic P Systems

The cell may be seen a small dimension but complex and dynamical system limited by a membrane which separates the external world from the internal cellular machinery. Acting as an interface, this membrane selectively collects molecules from surroundings of the cell and expels others that were produced or refused inside it.

The process of material exchange interfaced by the membrane has a parallel with systems' theory, in which the cell works as a box (isolated system) for transformation of substances, suggesting the existence of a computational process inside it. Gheorge Păun, aware of this mechanism, proposed a computational model called P system [14] (precursor of *membrane computing*) which is based on membrane interaction in cell systems but, at the same time, mathematically formal and consistent, using the concepts of multiset and rewriting systems for the construction of its formal framework.

Although full of new and interesting ideas for biological modeling, P system still presents mechanism too tied to formal languages that forbids its usage to model real-world metabolic and intra-cellular interaction [10, p.64]. Attentive to the necessities of bio-modeling, a new membrane computing computational model based on this system which is named *Metabolic P system* or, for short, *MP system* [10] [9, Chapter 3].

The primary goal of the MP system is to deterministically model metabolic processes, serving as a powerful (discrete) mathematical tool for expressing and supporting biological studies in the cell magnifying level; also, it meant to be a computational “intermediate language” for easy simulation of the formalized models; at last, it should use promptly understandable notation for potential users unfamiliar with the theoretical computer science jargon commonly found in new computational models.

Strongly influenced by chemical reactions, MP system has a reaction-like notation—that can be seen, also, as a *formal grammar* one—supported by recurrence equations and shifts the focus from pointwise string rewriting to a population transformation through the usage of conventional mole concept (as in chemistry). By the other side, its dynamics is supported by linear algebra and relies on matrix operations for solving the recurrence equation system that

characterizes the MP system as a (discrete) dynamical one. In technical jargon, an MP system can be mathematically represented using the support of a kind of formal grammar (named *MP grammar*). As a grammar object, the MP grammar defines all the rules of an MP system, including the process of multisets (through rules and functions), the elements allowed in the multisets and the initial state of the systems [9, p. 108].

Definition 3 (MP grammar). *An MP grammar G is a generative grammar for time series defined as*

$$G = (M, R, I, \Phi)$$

where:

1. $M = \{x_1, x_2, \dots, x_n\}$ the finite set of substances (or metabolites), and $n \in \mathbb{N}$ the quantities of substances.
2. $R = \{\alpha_j \rightarrow \beta_j \mid 1 \leq j \leq m\}$ the set of rules (or reactions), with α_j and β_j multisets over M , and $m \in \mathbb{N}$ the number of reactions.
3. $I = (x_1[0], x_2[0], \dots, x_n[0])$ is the vector of initial values of substances or the metabolic state at initial step (step 0).
4. $\Phi = \{\varphi_1, \varphi_2, \dots, \varphi_m\}$ is a set of functions (also called regulators), in which every $\varphi_j : \mathbb{R}^n \mapsto \mathbb{M}$, for $1 \leq j \leq m$, is associated with a rule $r_j \in R$.

According to definition 3, G generates (a set of) time series, each of them representing the “amount of quantity” of the substances during the time and its time series is calculated for any time t if and only if $\frac{t}{\tau} \in \mathbb{N}$ for a given constant τ .

Notwithstanding, the rules $\alpha_i \rightarrow \beta_i \in R$ depends, as equivalently happens in chemical reactions, on the quantities of the “substances” in the system, which can be expressed with the support of two different concepts: one that maps the multiplicity expressed in the rule for a substance to the actual number of molecules (of the substance) in the system, and; the quantity of mass the unit of the multiplicity represents (for a particular substance).

Hence, if the aforementioned three restrictions are that in consideration along with an MP grammar G , it is formally defines a *MP system*.

Definition 4 (MP system). *A MP system M is a discrete dynamical defined as*

$$\mathcal{M} = (G, \tau, \nu, \mu)$$

with

1. G being an MP grammar following the definition 3;
2. $\tau \in \mathbb{R}$, the period (amount of time) of a computational step;
3. $\nu \in \mathbb{R}$, the number of molecules that represents the (conventional) mole in the system;
4. $\mu \in \mathbb{R}^n$ is the vector of the mole masses of substances.

As described in definition 4, the quantities of the substances are dependent of its previous values and a variational function that may depend on other substances and parameters (in the case of the parametric MP system). This additional variance through time is represented as a recurrent equation which the future value of a substance X is represented as $x[i + 1] \propto x[i]$.

For the computation of these step values, nonetheless, two mathematical accessories were developed. The first, the *stoichiometric matrix*, is based on the arithmetic executed over chemical reactions to calculate the balance of molecules in a chemical system; the other, *equational metabolic algorithm*, synthesizes the whole computational process specified by the an MP system.

Definition 5 (Stoichiometric matrix). *Let $r_i = \alpha_i \rightarrow \beta_i$, where α_i (with an equivalent for β_i) is represented as $\sum k_{i,j}^+ \times X_j \mid k_{i,j} \in \mathbb{N} \wedge X_j \in M$.*

Let $\text{mult}^+(X_j, r_i) = k_{i,j}^+$ be the multiplicity, for the right side (α_i) of the rule r_i , of the substance X_j in the rule. Similarly, there is $\text{mult}^-(X_j, r_i) = k_{i,j}^-$ for the left side (β_i) of the rule.

A stoichiometric matrix \mathbb{A} , of dimension $|M| \times |R|$, has each of its elements defined by

$$a_{l,m} = \text{mult}^+(X_l, r_m) - \text{mult}^-(X_l, r_m)$$

with $1 \leq l \leq |M|$ and $1 \leq m \leq |R|$.

Definition 6 (Equational metabolic algorithm—EMA). *Let $U[i] = (\varphi_1(i), \varphi_2(i), \dots, \varphi_m(i))^T$ be the vector of values, in the time step i , of all regulators, and \mathbb{A} the stoichiometric matrix.*

The vector of substance variation at step i , $\Delta[i]$, is computed by the equation

$$\Delta[i] = \mathbb{A} \times U[i]$$

so-called Equational Metabolic Algorithm whom computes the value of any substance in the time future time step $i + 1$ through the recurrent equation

$$X[i + 1] = X[i] + \Delta[i]$$

The above definitions, now, complete specifies the discrete dynamical system and the ways to compute its values.

3.1 Positively Controlled Metabolic P System

The cell, motto of the MP systems, keep itself working based on metabolic rules. Those, of biochemical equations nature, require enough quantity the metabolites on the left-hand side of their equations to transform the matter and, hence, keep the metabolic circuitry active. Thus, if enough quantity of all metabolites are provided for all equations, the metabolic circuitry works properly; in the total absence, it does not work. However, in the case which a subset of rules do not have enough quantity of some metabolites, these rules are inactivated while the others keep their activities normally. As a result, the metabolic circuitry of the cell keeps working with a broken chain of reactions.

This kind of operation on cell metabolism happens because the (mathematical) operations in cells (and bio-chemistry, in general) happens in the space of

natural numbers, which implies that *half (decimal) quantities* or *debt (negative) quantities* cannot be considered in the (left-hand side of the) rules.

In the MP systems world, the above cell-like behavior is defined in a subclass called *Positively Controlled Metabolic P* systems, or *MPPC*: it is a standard MP systems with few controls to ensure all operations are executed in the set of positive numbers.

Definition 7 (MPPC Grammar). *A MPPC grammar \mathcal{G}_+ is a standard MP grammar \mathcal{G} respecting the following restrictions, at each computational step s_i :*

1. $\varphi|_{s_i} = \begin{cases} \kappa & , \text{ if } \kappa \geq 0 \\ 0 & , \text{ otherwise} \end{cases}$, for all $\varphi \in \Phi$;
2. $\sum_{\varphi|_{s_i} \in \Phi_x^-} \varphi \leq x$, with Φ_x^- the set of all fluxes related to rules in which the metabolite x is in the left-hand side of the rule; otherwise, $\varphi = 0, \forall \varphi \in \Phi_x^-$ at the execution step.

As it will be seen later in the next sections, this definition is useful not only for cell modeling, but also for establishing a comparison with universally computational devices.

4 Computational Architecture as Metabolic Systems

One of the goals of synthetic biology is to produce programmable metabolic systems that could, just as a designed machine, perform well-defined tasks for a certain objective. Some successful approaches of computer-aided design of biological systems already exists [1], but they are limited by the expressiveness of digital gates.

Theoretically, nonetheless, we are possible to synthesize metabolic systems more significant (in computational power) than those of Boolean circuits. For the present purpose, a programmable and Turing-potent (register) machine is implemented in a metabolic P system.

Theorem 1 (Translation of Register Machine to Positively Controlled MP grammar). *For any register machine \mathcal{R} exists an equivalent positively controlled MP grammar \mathcal{G}_+ .*

Proof of Theorem 1. Given a register machine $\mathcal{R} = (R, I, P)$ with $|R| = r$ and $|P| = p$, a positively controlled MP grammar $\mathcal{G}_+ = (M, Ru, I, \Phi)$ is constructed

1. adding a metabolite R_i in the set M for each register $R_i \in R$;
2. adding a metabolite I_j in the set M for each of the instructions in $I_j \in P$;
3. adding a metabolite L_j in the set M for each instruction $I_j \in P$ of the type JNZ;
4. adding a *HALT* metabolite in the set M ;

5. defining the initial state of the metabolites R_j equal to the initial values of the registers R_j , the initial values of all the other metabolites to 0 and the initial value of I_1 to 1;
6. adding the rules to Ru and the fluxes to Φ according to the following rules:
 - (a) if I_j is INC or DEC, then $I_j \rightarrow I_{j+1} : I_j$;
 - (b) if I_j is INC(R_i), then $\emptyset \rightarrow R_i : I_j$;
 - (c) if I_j is DEC(R_i), then $R_i \rightarrow \emptyset : I_j$;
 - (d) if I_j is HALT, then $I_j \rightarrow HALT : I_j$;
 - (e) if I_j is JNZ(R_i, I_k), then
 - i. $I_j \rightarrow L_j : I_j$;
 - ii. $L_j \rightarrow I_k : L_j - I_{j+1}$;
 - iii. $L_j \rightarrow \emptyset : I_{j+1}$; and,
 - iv. $\emptyset \rightarrow I_{j+1} : I_j - R_i$.

From the rules above, it is possible to notice that I_j and L_j instructions controls the execution flow of the system and satisfies

$$HALT + \sum_{j=1}^p I_j = 1$$

$$0 \leq \sum_{L_j \in M} L_j \leq 1$$

ensuring no two instructions are executed at the same time, but its execution starts from instruction I_1 and proceeds sequentially (or jumps to another one in case of a satisfying JNZ instruction).

All operations are mappings from and to the \mathbb{N} set once both \mathcal{R} and \mathcal{G}_+ , by definition, restrict their operations to this set.

At last, when a rule $I_j \rightarrow HALT$ is performed, the system is stuck in a fixed point since there is no rules for “exiting” this state. \square

5 Translation Strategy

Theorem 1 defines the relation between register machine and positively controlled metabolic P system, but it does not develop the intuition for its reasons. For a complete comprehension of this equivalence and, consequently, its impact in correlated fields, this section will focus on the constructive analysis of a MPPC from a register machine (just as proposed by Theorem 1) and discuss the challenges of this procedure.

5.1 Is It Possible To Translate Register Machines Into MP Systems?

A register machine is a general purpose computational device presenting the same computational power of a Turing machine; it allows the implementation of any algorithm and can deal with languages in the whole spectrum of the Chomsky hierarchy [8, p. 272]. By the other side, MP systems have been show

to be equivalent to Petri networks [3] and have been used to model complex systems, from (originally) metabolic systems to mathematical series [9, Chapter 3]; however, MP systems are not proved to be as powerful as a register machine and an equivalence between these systems cannot be set yet.

Nevertheless, MP systems presents a series of characteristics that induces to the idea of an existing equivalence between both systems. It is important to note that these features, by themselves, do not ensure the existence of the any kind of relation between the formalisms, but solely encourages the investigation through a number of theoretical arguments.

At first, *MP systems are (a particular kind of feedback) dynamical systems*; therefore, it receives a set of states, manipulates them and feeds them back to itself in order to keep the computation going on. It is a computational device with proved restricted power [3], but with potential for increasing it [7] [18] particularly because it does not restrict the set of functions it may works on—*i.e.*, the fluxes may be any computable function according to its definition.

Then, both systems works in a discrete representation of time, computing the elements in well-defined steps instead of continuously. Each step executes a finite number of instructions in a finite amount of time and, more, with a finite representation (*i.e.*, there is no guarantee of infinite precision for every object computed). Although these limitations on time are not strong enough to ensure the equivalence (after all, both Turing machines and finite state machines are discrete but not equivalent), it prevents analyzes that goes beyond the Turing equivalence, named super-Turing [5] [18].

At last, MP systems have representations with origins on (or inspired by) formal languages (grammar), rewriting systems and graph theory (MP graph [9, p. 109 and Figure 3.1]). Not enough, it is part of membrane computing, a wide field with other devices as powerful as (or even more than) the Turing machine [2, p. 179] (with properties, though, that make them impracticable for daily basis use).

5.2 What Is The Strategy To Implement?

In order to show that MP systems are general purpose computational devices as powerful as register machines (and, hence, Turing machines), it is enough to show that it is possible to implement in MP systems the same algorithms accepted by the register machine. And to satisfy the generality of these implementations, it suffices to translate the register machine instructions into MP equivalents and show they work properly for a well-defined subset of all acceptable algorithms in that machine.

Expressions of behavior in MP systems are described solely through its MP grammar [9, Definition 3.1], which is defined as a quadruple consisting of sets of metabolites, rules, initial states and regulators (or fluxes). Therefore, the definition of the four elements of a MP grammar defines univocally the operational actions of a system; if the definition is relaxed and the initial states are not specified, the relaxed MP grammar defines a whole class of behaviors instead of a specific one.

The strategy, then, is to restrict the definition of a *relaxed MP grammar that models the whole class of problems accepted by a register machine*, populating the sets that compose it according to the next strategies.

5.2.1 Metabolites Set

The metabolites set M in a MP system is the equivalent to $M = X \cup Y$, the union of the state space X and output value space Y of dynamical systems [7]. It declares all the variables (states) of the system, those allowed to change its values through the computation process.

Analyzing the register machine instructions [17, Section 4, p. 225] [12, Chapter 11] from Section 2 (the unique procedures that alters the states of a register machine), it is possible to recognize and isolate all the variables of the machine just looking to the addressed properties subject to manipulation. These can be classified as

- *Registers.* Three out of the four instructions (INC, DEC and JNZ) refers to registers, either for manipulation or query the data. Registers are the external variables [7, pp. 74] of the register machine, the ones of interest for an user that feeds the machine with a program.
- *Instruction Pointer.* Each of the instructions of the machine are executed sequentially, one at each computational step; however, this serial behavior can be modified with the usage of the JNZ instruction, which “jumps” the next execution step to another indexed instruction if an addressed register has its contents different from zero. And in order to keep the track of the instructions to be performed, there is a special kind of register (called *program counter*) whose content is the index of the instruction to be performed. Normally, its initial value is one and it is automatically incremented by one at the end of the execution of the instruction; however, JNZ is able to change it to whatever (valid) index it is necessary, hence qualifying program counter as a register machine’s variable.
- *Final State.* To indicate the end of a computational process executed by a register machine, there is a special instruction called HALT which signals the machine to stop its execution because the algorithm in process has finished its operation. Hence, *halt* (*final* or *acceptance state* [8]) is a particular that marks the end of the computational process.

5.2.2 Rules Set

The rules in a MP systems define the relation between the diverse variables of the system, without pay attention to the value update of the variables. Rules, then, characterizes the operation of the system and are presented in $\alpha \rightarrow \beta$ structure—which can be read as α *becomes* (or *transforms itself into*) β .

In the current scope, the definition of the structural behavior of the general MP system that simulates a register machine is done through the observation of both the register machine and the metabolite set: the former is detailed studied through the standpoint of the instruction set and the possible program flows (flowchart) and describes the behavior being reproduced in the new system; the latter provides the description of all the systems’ states (variables), the principal constituent of the rules and defines the three subdivisions of behavior of the register machine: *manipulation of the registers*, *manipulation of the flux of the instructions’ execution* and *final state of the computational process*.

Manipulation of the Registers The INC and DEC instructions are dedicated to manipulation of the registers, while JNZ uniquely consults their values.

The instructions that alters the values of the registers, consequently, are those to induce changes in (some of) the system's states (the *external* ones [7, p. 74]), expressing one of the (main) behavior of the system; therefore, they are modeled in the MP system through particular but very simple rules: they just add to or remove values from addressed registers.

Table 1: MP rules for register machine instructions that manipulate registers' values.

Register Machine Instruction	MP Rule
INC(R_i)	$\emptyset \rightarrow R_i$
DEC(R_i)	$R_i \rightarrow \emptyset$

The first of the rules of Table 1 can be interpreted as “from somewhere the register R_i receives a value” (and the second one as “the value of the register R_i goes to somewhere”) because of the structure of MP rules: inside a delimited environment (*e.g.*, a cell), the *principle of mass conservation* must be preserved and any addition (or removal) of metabolites quantities that breaks this principle must “come from (or goes to) somewhere”; *somewhere*, in this context, must be understood as *outside the membrane* delimiting the MP system.

Manipulation of the Flux of Instructions Not all the states of system are registers; those responsible for the sequential execution of the system and the control of its computational workflow, the *instruction pointers*, represents the computational machinery and hide the trickiest translation process between register machine instructions and MP rules. The evaluation of these variables may be divided in two different subclasses: *sequential execution of instructions* and *manipulation of the execution's workflow*.

The first of the subclasses, *sequential execution of instructions*, consists of situations when the token of active instruction is passed from one instruction to another consecutive. Thus, let $I = \{I_1, I_2, \dots\}$ be the set of indexed instructions of a register machine representing an algorithm. The computation of the algorithm represented by I is defined as the sequential execution $I_1 \vdash I_2 \vdash \dots$, if and only if I_j is a instruction of the type INC or DEC.¹ Then, from the previous statement, it is trivial to define the following conversion rule:

Let I_j, I_{j+1} be the j^{th} and $(j+1)^{\text{th}}$ indexed instructions, with I_j belonging to one of the types INC or DEC. Then, there is a MP rule $I_j \rightarrow I_{j+1}$ that sequentially executes the instruction I_j in a MP system.

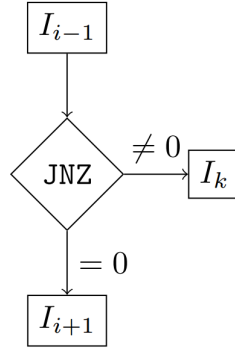
For the above rule, it is important to notice that there is the restriction of the type of instruction (INC or DEC) solely for the I_j instruction; the I_{j+1} , on the other hand, can be any of the register machine instructions.

¹Although JNZ instruction also pass the token in a sequential manner, it is not true for every performance of it; therefore, this instruction is excluded of this subclass and is analyzed later.

The other subclass, manipulation of the execution's workflow, comprises the situations in which the computation of I does not follow the sequential disposition of the instructions, but changes the execution order under certain circumstances. For this, it is required the JNZ instruction, the exclusive instruction capable of performing such re-arrangements.

Dissecting the JNZ instruction (Definition 2), it is possible to see it has two arguments, a register address and an instruction address, one used to perform an inequality comparison against zero of a register value and the other to redirect the current workflow in case the comparison results are true. *i.e.* if the addressed register R_i does not have its value equal to zero; since this instruction is inherently composed of two operations, it persuades us to convert it in, at least, two MP rules.

Figure 1: Flowchart of the JNZ(R_i, I_k) instruction.



At first, let us focus in the *comparison* operation of JNZ. Looking at Figure 1, it is easy to see the detour of the sequential instructions' execution happens when the value of the register R_i differs from zero; this remark suggests most of the trickery will be developed in order to make this execution branch feasible in MP systems.

The artifice is to create a metabolite L_j that represents the virtual comparison instruction of instruction $I_j \equiv \text{JNZ}(R_i, I_k)$ and, then, let it support the control of the redirection of the instructions' execution.

Thus, L_j must receive the token previously in I_j and, in your own computational step, choose whether I_{j+1} or I_k will be the next instruction to be carried out.

For the reception of the token, it follows the same mechanism previously used for sequential execution of the instructions: the I_j token becomes the L_j token with the simple rule $I_j \rightarrow L_j$.

The choice of the instruction path, in turn, makes extensive use of *regulators* (which will be discussed ahead in Section 5.2.3) in order to produce its rules. Intuitively, though, it is possible to develop them.

When the contents of $R_i \neq 0$, L_j must redirect the execution flow to the instruction I_k through the transfer of its token to this new instruction; therefore, again, the sequential rule style operates and the produced rule is $L_j \rightarrow I_k$.

However, when $R_i = 0$, a problem arises: neither $I_j \rightarrow I_{j+1}$ nor $L_j \rightarrow I_{j+1}$ rules may be used to give, in the context of *a metabolite becomes another metabolite*, I_{j+1} the execution token—the former (I_j) does not have tokens

anymore and the latter (that would be the correct rule) was used to *transform* the token from L_j to I_k . The solution, then, is to create two rules to reproduce the behavior of $L_j \rightarrow I_{j+1}$: one takes the token from L_j and the other gives it to I_{j+1} .

$$L_j \rightarrow I_{j+1} \Rightarrow \begin{cases} L_j \rightarrow \emptyset \\ \emptyset \rightarrow I_{j+1} \end{cases}$$

At end, an instruction $I_j \equiv \text{JNZ}(\mathbf{R}_i, \mathbf{I}_k)$ becomes four MP rules summarized in Table 2.

Table 2: MP rules for the $I_j \equiv \text{JNZ}(\mathbf{R}_i, \mathbf{I}_k)$.

Functional Description	MP Rule
Pass token to <i>pointer of virtual comparison instruction</i> L_j	$I_j \rightarrow L_j$
If $R_i \neq 0$, pass token to I_k	$L_j \rightarrow I_k$
If $R_i = 0$, pass token to I_{j+1}	$L_j \rightarrow \emptyset$ $\emptyset \rightarrow I_{j+1}$

Final State of the Computational Process At the end of a computational process, some kind of flag is necessary to signalize its termination, otherwise there is no way to guarantee we are seeing a partial result of the computation or the system's final state. Turing machines, in their formal definition [8, Definition 4.1.1], make this explicit signalization through the *halting states* set, and the register machine uses its **HALT** instruction; these halting objects, as the name suggests, stops the computational device execution.

By the other hand, MP systems are feedback discrete dynamical systems and the concept of stopping the computation does not exist in this context. In order to imitate this behavior, it is possible to add to the systems' variables an additional state (which we will conveniently call it *halt state*) which marks the end of the systems' computational process and, at the same time, is a *fixed point* of the dynamical system. For this purpose, it is enough the addition of a single halt state that is the destination of the token from the **HALT** instructions and the absence of rules with the halt state in the left-hand side of it. Hence,

Let I_j be a **HALT** instruction. Then, there is a MP rule $I_j \rightarrow \text{HALT}$ that models the end of the computational process with signalization of the final state.

Clarifying the above definition, a register machine does not restricted your programs to have an unique **HALT** instruction and, thus, it may have several of them, all of them going to a single halt state. Therefore, the "token" of the j^{th} instruction *becomes* the signal of end of computation and no other instruction is executed, because: (i) the value ("token") that controls the instruction flow is passed from the instructions (I_j) to the halt state (**HALT**) and (ii) there is no rules in which the halt state is consumed (*i.e.*, in the left-hand side).

In fact, those two arguments also justify the halt of the device as a fixed point of the dynamical systems: with no rules being executed, no modification in the states are performed and, that being so, any other execution step of the MP system produces the same, previous state ($S_{halt} \vdash S_{halt} \vdash \dots \vdash S_{halt}$).

5.2.3 Regulators Set

In a MP system, the metabolites set defines the state variables of the system. The rules set, its skeleton behavior. But the update of the values of the state variables depends on mathematical functions associated with the rules, also known as *regulators*.

The regulators computes the rates in which the metabolites in the left-hand side of rules are transformed into the ones in the right-hand side. In both mathematics and biology, regulators are often called *flux*.

Since does not make sense to study regulators detached from rules, the classification of the equivalent regulators of a register machine will follow similar subdivision used for the rules: *update of the registers' values*, *update of the instruction pointers* and *signalization of end of computation*.

Update of the Registers' Values The rules presented in the Table 1 describes the process of updating a register value, but not the functions which actualizes it; these functions (the regulators) can be easily inferred from the specification of the register machine instructions.

Initially, the INC operation: it is a primitive recursive function which adds a single unit to the actual value, or $\text{INC}(R_i) = R_i + 1$, which can also be stated, in a regulator perspective, as “INC(R_i) changes the value of R_i with the constant rate of 1”. The latter declaration, in turn, denotes the value of regulator as the constant 1; then, it is easy to define the regular for the INC instruction: $\varphi_{\text{INC}} = 1$.

Conversely, DEC follows a similar reasoning, $\text{DEC}(R_i) = R_i - 1^2$, which can also be stated as “DEC(R_i) changes the value of R_i with the constant rate of -1 ”. Nonetheless, it is also necessary to notice the DEC rule “consumes” the R_i value is (because R_i is in the left-hand side of $R_i \rightarrow \emptyset$) and, hence, its flux is already negative. Thus, $\varphi_{\text{DEC}} = 1$.

Update of Instruction Pointers and Final State As with the case of the update of the registers' values, the behavior of the instruction pointers are well defined in the Section 5.2.2, but no modification to the state representing the instruction pointers are yet defined or performed. But before defining the regulators for their rules, it is important to reflect a little about the nature of the instruction pointers.

The instruction pointers states are flags that, when activated, indicates which of the instructions are being processed in the current computational step; they are the stratagem used to transform the parallel execution of rules in MP system (matrix multiplication at Definition 6) into sequential ones as present in the register machines.

²Actually, $\text{DEC}(R_i) = \max\{R_i - 1, 0\}$ since it is an operation $\mathbb{N} \mapsto \mathbb{N}$. It is guaranteed by the *positive control* of MPPC systems and will be discussed, again, further in this section.

The idea of the instruction pointers are similar to the *program counter* of counter and random access machines, with the particularity that each instruction has its own instruction pointer, they are limited to binary values (deactivated and activated, respectively 0 and 1) and no two instruction pointers may be activated at the same time, formally defined as³ $Halt + \sum I_j = 1$.

Thus, it is easy to define the regulators of the rules for sequential execution of instructions: for each rule of the form $I_j \rightarrow I_{j+1}$ or $I_j \rightarrow Halt$, the regulator is defined as $\varphi_{pointer} = I_j$.

(The *Halt* state variable may be seen as a particular case of the instruction pointer ones: it marks the end of the computational process instead of the instruction being computed.)

Finally, we must carefully design the regulators for the previously defined $I_j \equiv JNZ(R_i, I_k)$ rules. Thus, let us recall the mechanism of this function:

it compares the value of the register R_i against the number zero and if it is equal, the next instruction to be executed will be the sequential instruction I_{j+1} , otherwise it will re-route the execution flow to the instruction I_k .

At first, then, JNZ delegates to the virtual comparison instruction L_j the comparison of the register R_i value against zero. Modeled by the rule $I_j \rightarrow L_j$, it is actually a particular kind of sequential instruction execution, but *inside* the JNZ rule. For this reason, its regulator follows the previous defined for the sequential execution rules and assumes the value of I_j .

Next, there are two rules for the case in which $R_i = 0$: one passes the token to the sequential instruction I_{j+1} and the other certifies the redirection to the I_k instruction will not ever occur. The former case, represented by the rule $\emptyset \rightarrow I_{j+1}$, is a variation of the sequential execution when $R_i = 0$, what drives us to consider I_j (sequential execution) and R_i (register value) as elements for the regulator. Observing the Table 3 of the possible values of these metabolites, it is easy to infer its regulator as $\varphi_{\emptyset \rightarrow I_{j+1}} = \max(I_j - R_i, 0)$.

Table 3: Values of I_{j+1} depending on the values of $I_j \equiv JNZ(R_i, I_k)$ and R_i .

I_j	R_i	I_{j+1}
0	0	0
0	$\neq 0$	0
1	0	1
1	$\neq 0$	0

For the latter of the rules when $R_i = 0$, $L_j \rightarrow \emptyset$, its regulator just have to control that the virtual comparison instruction L_j will not be active (*i.e.*, its value will be zero) when I_{j+1} is; given that L_j is solely activated by the j^{th} instruction by a sequential-like rule, it is certainty it will not get any other increase of value outside the JNZ workflow and, hence, it has a maximum value of one. Then, it is trivial to infer that $\varphi_{L_j \rightarrow \emptyset} = I_{j+1}$, establishing I_{j+1} as the repressor metabolite of L_j .

³The presence of *Halt* is justified by the necessity of the system be in execution mode, not halted.

Finally, the lasting rule of JNZ concerns to the re-route of the execution flow to the k^{th} instruction of the algorithm, I_k . As previously state, a redirection of the flux of execution happens *when the value of the register R_i is different of zero, i.e.*, it is a “secondary level” task inside the JNZ workflow, which depends on the rules when $R_i = 0$. By the other hand, the redirection happens, as state by its rule $L_j \rightarrow I_k$, as a consequence of the activation of the virtual comparison instruction. Minimizing the metabolites set that influences the re-route sub-task of JNZ and combining all their possible values in the Table 4, the pattern that arises between this table and Table 3 correctly induces to conclude the regulator for this last rule is $\varphi_{L_j \rightarrow I_k} = \max\{L_j - I_{j+1}, 0\}$.

Table 4: Values of I_k depending on the values of L_j and I_{j+1} .

L_j	I_{j+1}	I_k
0	0	0
0	1	0
1	0	1
1	1	0

It is interesting to notice, however, that both $\emptyset \rightarrow I_{j+1}$ and $L_j \rightarrow I_k$ have regulators of the form $\max\{X - Y, 0\}$, or subtraction in the natural set of the numbers, in order to restrict the regulators to become negative numbers. This guard, also present in the DEC instruction, provides two curious features to the system:

1. the comparison operator (*greater-than*) required in JNZ, ensuring a positive, not null regulator if and only if $X > Y$; and,
2. preservation of the coherence of the rules.

While the former characteristic is evident and does not require further explanation, the latter preserves the structural rules of the system: a negative regulator inverses the behavior of the associated rule, *i.e.*

$$(X \rightarrow Y : -\epsilon) \equiv (Y \rightarrow X : \epsilon)$$

Hence, the concession of negative fluxes in the system would allow the modification of its set of the rules, with the modification of a certain rule $r_i = \{X \rightarrow Y, Y \rightarrow X\}$ depending of its regulator—or, in other words, the system could represent two different programs depending of the positiveness of the regular of r_i .

Furthermore, this restriction pattern goes in accordance with two other (and important) principles: the *positive control* 7 of MP grammars, which asserts that fluxes must be greater or equal to zero, and *computability over the natural numbers* [8, Definition 4.4.1, Figure 4-19 and Section 4.7] [17, Section 2].

5.2.4 Program and the Initial States

In Section 5.2 we introduced the concept of *relaxed MP grammar* as a variation of the MP grammar definition without the requirements to set the initial states.

This flexibility allowed to construct the equivalence instruction of register machine in MP as general as it is possible, without restraining the systems to a single, particular behavior: a relaxed grammar defines a whole class of dynamics based on general rules, not on specific simulations dependent of initial states values.

Although it is enough to define an equivalence with register (and Turing) machines through bijection of its composing elements, relaxed MP grammar does not allow the definition of the concept of *program* [8, pp. 210–211 and Definition 4.4.1] [12, p. 202] because nothing such as *initial configuration* [8, Definition 4.4.2] is specified.

The initial configuration solely defines values for the states of the system and no modification of the designed model is undergone. It acts as a constrainer of the device model to ensure the correct execution of the dynamics.

Reviewing the states (metabolites) set of the MP system equivalent to the register machine, Section 5.2.1 and later addition in Section 5.2.2, it is possible to partition them under two categories: (i) the *memory units*, which comprise the registers-equivalent metabolites (R_i); and, (ii) the *instruction pointers*, containing the instruction pointers I_j , the halting state $Halt$ and the virtual comparison instruction L_j . The first kind is, as in similar computational models [7, Section 2.1.1] [12, Section 11.1] [8, Section 4.4], memory units that keeps the state of the internal computations but also serves as an interface for input (and, at the end of the computation process, output) data in the program; thus, it must always be initialized with zero values when representing internal computation states, while is open to be freely set to (positive) values when representing storage for input data.

The second kind, instruction pointers, are internal state that control the execution of the MP system as a sequential, computational device. The instruction pointers and virtual comparison instructions, together, represent the program counter [8, Definition 4.4.1] in the MP formalism while the halt state indicates the end of the computation.

In order to correctly execute a program from a register machine, nonetheless, it is also necessary to specify the states of the instruction pointers satisfying the two following constraints:

$$I_1 = 1 \tag{1}$$

$$\sum_{i=1}^{|\mathcal{K}|} K_i = 1, \text{ for any } t \in \mathbb{N} \text{ and } K_i \in \mathcal{K} = I_j \cup L_j \cup Halt \tag{2}$$

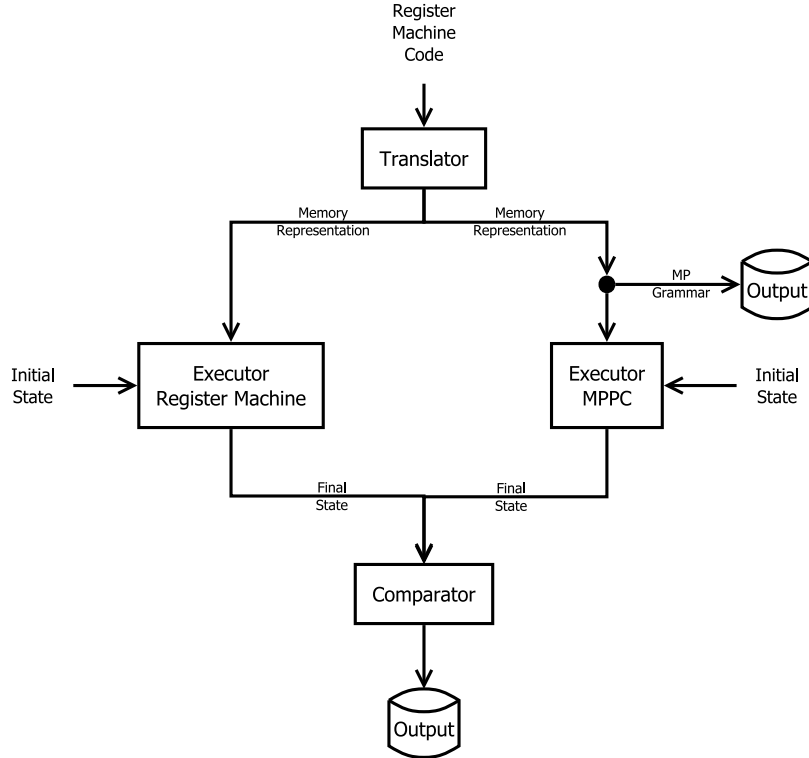
Equation (1) defines a sequence $S = (I_1, I_2, \dots)$ for the instructions codified as MP rules merely with the definition of its first element, while the other sequential elements follows from the sequential execution rules (Section 5.2.2). Equation (2), in turn, restricts the execution to solely one instruction (pointer) at each step, guarantees the device is either on work or halted and, finally, forces all the states in the instruction pointer category to have initial values equal to zero (except for $I_1 = 1$).

6 Software

Following the theoretical development that derived the Theorem 1, a piece of software were codified in order to provide experimental evidence of the validity of the theory, as well as a tool for automatic translation of register machines specifications into MP systems—which, later, expanded for also simulated both systems and ensure their equivalence. Here, though, the bisimulation is not verified, but instead a verification of the final state of both systems is performed.⁴ However, if one of the systems does not halt, it is not possible to perform the comparison of the systems, a expect situation and in accordance to the *halting problem* [8, Section 5.3] [19, Section 5.1].

The existing software, a command-line tool available for the major operating systems players, can be divided in three main parts: *translator*, *executor* and *comparator*, as represented in Figure 2. This modular structure allows an unique flexible feature to the tool to choose the atomic operations to perform for a determined task; also, it provides the opportunity to implement parallel execution of modules and easy addition of new components.

Figure 2: Dataflow of the software.



When called by the command-line, the software may solely translate the register machine specification into a valid MP grammar (Figure 4) or generate

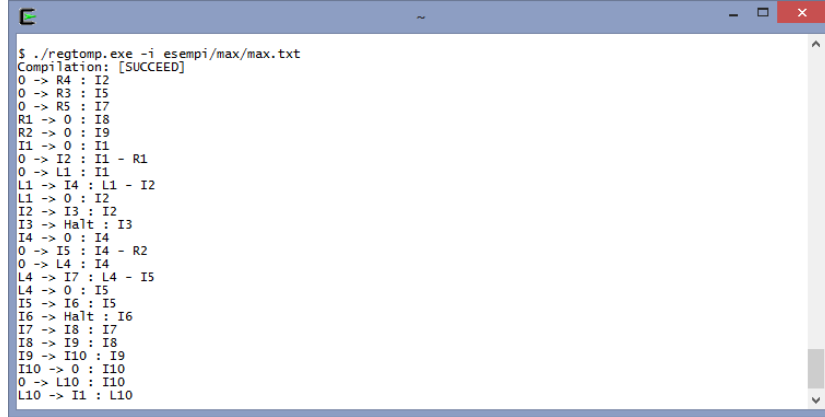
⁴The bisimulation is not necessary to be performed in the software due to the result of the Theorem 1 which states an equivalence. There is space, though, for this verification if the intermediate steps of both simulations are compared according to [15, Definition 1.4.2].

the equivalent MPPC system to the register machine, simulate both of them and print if they are equivalent based in two criteria: both system halted and the final state of the registers and their respective metabolites in the MP grammar have the same values (Figure 5).

Figure 3: Register machine specification for the $\max(R_1, R_2)$ function.

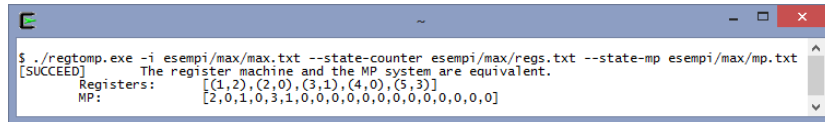
```
JNZ(1, 4)
INC(4)
HALT
JNZ(2, 7)
INC(3)
HALT
INC(5)
DEC(1)
DEC(2)
JNZ(5, 1)
```

Figure 4: Translation from register machine to MP grammar with the command-line tool.



```
$ ./regtomp.exe -i esempi/max/max.txt
Compilation: [SUCCEED]
0 -> R4 : I2
0 -> R3 : I5
0 -> R5 : I7
R1 -> 0 : I8
R2 -> 0 : I9
I1 -> 0 : I1
0 -> I2 : I1 - R1
0 -> L1 : I1
L1 -> I4 : L1 - I2
L1 -> 0 : I2
I2 -> I3 : I2
I3 -> Halt : I3
I4 -> 0 : I4
0 -> I5 : I4 - R2
0 -> L4 : I4
L4 -> I7 : L4 - I5
L4 -> 0 : I5
I5 -> I6 : I5
I6 -> Halt : I6
I7 -> I8 : I7
I8 -> I9 : I8
I9 -> I10 : I9
I10 -> 0 : I10
0 -> L10 : I10
L10 -> I1 : L10
```

Figure 5: Simulation and comparison of register machine and equivalent MP grammar with the command-line tool.



```
$ ./regtomp.exe -i esempi/max/max.txt --state-counter esempi/max/regs.txt --state-mp esempi/max/mp.txt
[SUCCEED] The register machine and the MP system are equivalent.
Registers: [(1,2),(2,0),(3,1),(4,0),(5,3)]
MP: [2,0,1,0,3,1,0,0,0,0,0,0,0,0,0,0,0,0]
```

In the previous example, both devices perform the computation $(R_1, R_2, R_3, R_4, R_5) = (5, 3, 0, 0, 0) \vdash^* (2, 0, 1, 0, 3)$ equally, writing 1 at R_3 to indicate the value of the register $R_1 \geq R_2$ (otherwise, $R_4 = 1$ to indicate $R_1 < R_2$).

7 Conclusion

Among the several models of cells and metabolism, it is difficult to find one that naturally models the cellular system and provides significant insight to challenge the biological machinery against existing and well-known synthesis theory, such as digital circuits or computer programs. Thus, these models are, usually, very useful for the systems biology analysis, but they lack contribution for the counterpart synthetic biology.

The present work extends the existing Metabolic P system, extensively used for analyzing biological behavior, to computation theory and provides theoretical and software support to convert programmable instructions into positively controlled Metabolic P grammar. Particularly, it stands out from competing models because (i) MP grammar is the formalism for every metabolic process; (ii) it is a proven Turing-powerful model (under positively controlled restriction); (iii) MPPC mimics the cell behavior in the borderline situations; (iv) it provides software tools to transform software specification into MPPC description.

These features, when combined with hardware synthesis techniques [13] or database of metabolites (similar to [1]), easily brings the ambition of lab-on-chip [6] and synthesis biology to the laboratories day-by-day reality.

References

- [1] Jacob Beal, Ting Lu, and Ron Weiss, *Automatic compilation from high-level biologically-oriented programming language to genetic regulatory networks*, PLoS ONE **6** (2011).
- [2] Cristian Sorin Calude and Gheorghe Păun, *Bio-steps beyond Turing.*, Bio Systems **77** (November 2004), no. 1-3, 175–94.
- [3] Alberto Castellini, Giuditta Franco, and Vincenzo Manca, *Hybrid functional petri nets as MP systems*, Natural Computing **9** (2010), 61–81.
- [4] Barry S. Cooper, *Computability Theory*, Chapman and Hall/CRC, 2004.
- [5] Marian Gheorghe and Mike Stannett, *Membrane system models for super-Turing paradigms*, Natural computing, August 2012, pp. 253–259.
- [6] Lauren Gravitz, *Cell on a Chip*, 2009.
- [7] Diederich Hinrichsen and Anthony J. Pritchard, *Mathematical Systems Theory I: Modelling, State Space Analysis, Stability and Robustness*, Texts in Applied Mathematics, vol. 48, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.
- [8] Harry Lewis and Christos Papadimitriou, *Elements of the Theory of Computation*, 2nd ed., Prentice-Hall, Upper Saddle River, 1997.
- [9] Vincenzo Manca, *Infobiotics: Information in Biotic Systems*, Emergence, Complexity and Computation, vol. 3, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
- [10] Vincenzo Manca, Luca Bianco, and Federico Fontana, *Evolution and Oscillation in P Systems: Applications to Biological Phenomena*, Membrane computing se - 4, 2005, pp. 63–84.
- [11] Vincenzo Manca and Rosario Lombardo, *Computing with Multi-membranes*, Membrane computing, 2012, pp. 282–299.
- [12] Marvin Minsky, *Computation: Finite and Infinite Machines*, 1st ed., Prentice Hall, 1967.
- [13] Volnei A Pedroni, *Circuit Design with VHDL*, 1st ed., MIT Press, Cambridge, United States of America, 2004.
- [14] Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, *The Oxford Handbook of Membrane Computing* (Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, eds.), Oxford University Press, 2010.
- [15] Davide Sangiorgi, *Introduction to Bisimulation and Coinduction*, Cambridge University Press, 2012.

- [16] Rahul Sarpeshkar, *Ultra-Low Power Bioelectronics. 1*, 1st ed., Cambridge University Press, 2010.
- [17] J. C. Shepherdson and H. E. Sturgis, *Computability of Recursive Functions*, Journal of the ACM **10** (1963), 217–255.
- [18] Hava T. Siegelmann and Shmuel Fishman, *Analog computation with dynamical systems*, Physica D: Nonlinear Phenomena **October** (1998), 1–38.
- [19] Michael Sipser, *Introduction to the Theory of Computation*, 3rd ed., Cengage Learning, Boston, USA, 2012.